iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79556

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

An Agentic System for Code Review & Debugging

Mrs. P. Pravallika[1], B. Devisree[2], Ch. Rupasri[3], K. Devi Sushma sree[4]
[1]MTech, [2, 3, 4]B.Tech, Computer science & Engineering, Bapatla Women's Engineering College, Bapatla, AP, INDIA

Abstract: ***Automated code review is fundamental to software quality assurance, yet existing approaches either rely on deterministic static analysis tools that lack pedagogical depth or on large language models (LLMs) that risk generating unverified and hallucinated feedback. To address this limitation, this paper introduces an agentic AI architecture for Python code review that decomposes the review process into specialized, collaborative agents with distinct reasoning roles. The proposed framework employs a modular multi-agent pipeline consisting of: (i) a static analysis agent responsible for deterministic detection of syntactic errors, code smells, and security vulnerabilities; (ii) an interpretation agent that transforms validated diagnostic outputs into structured, beginner-oriented explanations using a grounded large language model; (iii) a scoring agent that quantitatively evaluates code quality and assigns skill levels; and (iv) a learning agent that generates adaptive practice recommendations based on identified knowledge gaps. By enforcing role separation and constraining generative reasoning to verified analytical outputs, the architecture reduces hallucination risk while preserving the expressiveness and instructional richness of LLM-based feedback. Unlike monolithic LLM-driven systems, the proposed agentic design ensures modularity, interpretability, reproducibility, and extensibility. Experimental evaluation on curated Python programming tasks demonstrates that the multi-agent collaboration model improves feedback reliability, conceptual clarity, and review efficiency compared to standalone static analysis tools and single-agent LLM baselines. The results highlight the potential of agent-oriented AI architectures for intelligent tutoring systems and next-generation AI-assisted software engineering workflows.***

Index Terms: Agentic AI, Multi-Agent Systems, Large Language Models (LLMs), Automated Code Review, Intelligent Tutoring Systems, Software Engineering, Static Analysis, Artificial Intelligence, Machine Learning, Natural Language Processing.

I. INTRODUCTION

Ensuring high code quality and software security remains a fundamental challenge in modern software engineering. As software systems grow in scale and complexity, maintaining readability, correctness, maintainability, and vulnerability resilience becomes increasingly difficult. Traditional peer-based code review processes serve as a critical quality assurance mechanism, enabling defect detection, adherence to coding standards, and security validation. However, manual reviews are labour-intensive, time-consuming, and inherently inconsistent, particularly in educational and large-scale collaborative environments.

Static analysis tools were introduced to partially automate this process by detecting syntactic errors, rule violations, and predefined vulnerability patterns. While such tools provide deterministic and reliable diagnostics, their feedback is often terse, technical, and lacking pedagogical explanation. Consequently, novice developers may struggle to understand the underlying causes of detected issues, limiting the learning value of automated review systems.

Recent advances in Large Language Models (LLMs) have introduced new possibilities for AI-assisted code review. LLMs demonstrate strong capabilities in code understanding, explanation generation, and natural language reasoning, enabling them to provide human-like feedback, refactoring suggestions, and conceptual clarifications. However, LLM-driven systems operate probabilistically and may generate hallucinated, contextually inaccurate, or unverified recommendations. When deployed as monolithic code reviewers, such systems raise concerns regarding reliability, reproducibility, and trustworthiness in professional and educational settings. To address these limitations, this paper proposes an agentic AI architecture for automated code review that decomposes the review workflow into specialized, collaborative agents with distinct reasoning responsibilities. Instead of relying on a single monolithic model, the proposed framework introduces a structured multi-agent pipeline comprising: (i) a static analysis agent responsible for deterministic code inspection; (ii) an interpretation agent that transforms verified diagnostics into structured, beginner-oriented explanations using a grounded LLM; (iii) a scoring agent that quantifies code quality and assigns skill levels; and (iv) a learning agent that generates adaptive improvement recommendations based on detected knowledge gaps.

This role-based decomposition enables controlled interaction between symbolic verification and generative reasoning. By constraining LLM outputs to validated static analysis findings, the system mitigates hallucination risks while preserving interpretability and instructional depth. Furthermore, the modular design enhances reproducibility, extensibility, and empirical evaluation, distinguishing the proposed approach from existing single-agent LLM-based code assistants.

©IJRASET: All Rights are Reserved

The contributions of this work are threefold:

1) Architectural Contribution: We introduce a modular agentic AI framework for code review that separates detection, interpretation, evaluation, and pedagogical recommendation into specialized agents.
2) Reliability Mechanism: We demonstrate how grounding generative feedback in deterministic static analysis reduces hallucinated suggestions and improves trustworthiness.
3) Educational Integration: We design a scoring and adaptive learning component that transforms automated review into an intelligent tutoring system for novice programmers.

Through experimental evaluation on curated Python programming tasks, we analyse feedback clarity, reliability, and review efficiency compared to standalone static analysis tools and unconstrained LLM baselines. The results demonstrate that agent-oriented AI architectures provide a principled pathway toward trustworthy, scalable, and pedagogically meaningful AI-assisted software engineering systems.

II. BACKGROUND AND RELATED WORK

The integration of artificial intelligence into software engineering has significantly transformed automated code analysis and review processes. This section reviews the evolution of traditional code review practices, static analysis techniques, large language model (LLM)-based approaches, and recent developments in agentic AI architectures.

A. Traditional Code Review Practices

Code review has long been an essential quality assurance mechanism in software development. In conventional workflows, developers manually inspect code changes submitted by peers to detect logical errors, ensure compliance with coding standards, and identify potential security vulnerabilities. Beyond defect detection, manual reviews foster collaborative learning and knowledge sharing within development teams.

Despite their benefits, manual code reviews exhibit several limitations. The process is time-consuming, particularly for large-scale systems, and its effectiveness depends heavily on reviewer expertise and attention. Human reviewers may overlook subtle errors under tight deadlines or when dealing with complex codebases. Additionally, review consistency varies across teams and projects, limiting scalability in modern development environments.

B. Static Analysis Tools

To address the limitations of manual inspection, automated static analysis tools were introduced. Static Application Security Testing (SAST) tools and linters analyse source code without execution, detecting syntax violations, code smells, and known vulnerability patterns. Tools such as SonarQube, ESLint, and Checkmarks have become standard components of continuous integration pipelines.

Static analysis tools offer deterministic and reproducible diagnostics. They are effective at identifying rule-based violations and predefined vulnerability signatures. However, their reliance on handcrafted rules limits contextual understanding. While they can detect *what* is wrong, they often fail to explain *why* the issue occurs or how it relates to broader design principles. As a result, their pedagogical value—particularly for novice developers—remains limited.

C. Large Language Models in Code Review

The introduction of transformer-based Large Language Models (LLMs) has opened new possibilities for intelligent code assistance. Models trained on extensive corpora of natural language and source code demonstrate capabilities in code completion, refactoring, automated repair, and explanation generation.

Prior research has shown that LLMs can assist developers by generating context-aware code suggestions and identifying non-trivial defects. For example, transformer-based models have been successfully applied to code synthesis and completion tasks [3]. Subsequent studies demonstrated that LLMs can evaluate and repair code snippets, achieving competitive performance in automated program repair benchmarks [4]. These capabilities highlight the potential of LLMs to enhance productivity and improve code quality.

However, LLM-based systems operate probabilistically and lack formal verification mechanisms. They may generate hallucinated issues, incorrect fixes, or contextually inappropriate recommendations. Furthermore, outputs may vary across runs due to non-deterministic inference. These limitations raise concerns regarding reliability, reproducibility, and trust when deploying LLMs in professional development workflows.

D. Agentic AI and Multi-Agent Architectures

Recent advances in agentic AI propose decomposing complex tasks into specialized agents that collaborate under structured coordination mechanisms. Instead of relying on a single monolithic model, agentic systems distribute responsibilities across role-specific components, enabling modularity, traceability, and controlled reasoning.

Multi-agent architectures have demonstrated advantages in reliability and interpretability by separating deterministic processing from generative reasoning. In such systems, information flow between agents can be explicitly constrained, reducing error propagation and improving system transparency. These properties are particularly relevant in safety-critical and verification-oriented applications.

Despite growing interest in agent-based AI systems, their application to automated code review remains relatively unexplored. Most existing AI-assisted coding tools rely either on static rule engines or standalone LLM inference without structured grounding.

E. Research Gap and Motivation

The existing literature reveals a fundamental trade-off between symbolic reliability and generative flexibility. Static analysis tools provide high precision but limited interpretability. LLM-based systems provide rich explanations but lack guaranteed correctness. Current approaches rarely integrate both paradigms within a structured architectural framework.

Furthermore, prior systems primarily focus on defect detection or code generation, with limited emphasis on educational support, skill evaluation, and adaptive learning guidance. This gap is particularly significant in academic and beginner-oriented programming environments.

The proposed work addresses these limitations by introducing a structured agentic architecture that integrates deterministic static analysis with grounded LLM-based explanation, quantitative scoring, and adaptive learning recommendations. By separating detection, interpretation, evaluation, and tutoring functions into specialized agents, the framework enhances reliability, interpretability, and pedagogical value

III. IMPLEMENTATION OF LLMS IN CODE REVIEW PROCESSES

The integration of Large Language Models (LLMs) into code review workflows introduces advanced capabilities for automated reasoning, contextual explanation generation, and developer assistance. However, deploying LLMs in professional and educational environments requires structured orchestration mechanisms to ensure reliability, interpretability, and controlled generative behaviour. This section presents the implementation strategy adopted in the proposed agentic multi-agent framework, emphasizing modular system design, grounding constraints, and workflow integration.

A. Role of LLMs Within the Agentic Architecture

In conventional AI-assisted development tools, LLMs are typically deployed as monolithic reviewers that directly analyse source code and generate feedback. Although expressive, such configurations may produce hallucinated findings, unsupported recommendations, or inconsistent outputs across executions.

In the proposed framework, the LLM operates as a specialized Interpretation Agent within a structured multi-agent pipeline. Its function is deliberately constrained to transforming verified static analysis diagnostics into structured natural-language explanations. The responsibilities of the Interpretation Agent include:

1) Translating validated diagnostic outputs into beginner-oriented explanations.
2) Providing contextual clarification of detected issues.
3) Suggesting corrective actions explicitly grounded in verified findings.
4) Generating structured improvement guidance without introducing unsupported claims.

Formally, the LLM component performs a conditional transformation defined as:

$$\mathbf{A_LLM : (D, P) \rightarrow E}$$

where P denotes the input source code, D represents verified diagnostics generated by the Static Analysis Agent, and E denotes structured explanatory feedback. By conditioning generative reasoning strictly on deterministic outputs, the architecture enforces grounding constraints that reduce hallucination risk and enhance interpretability.

B. System Architecture and Workflow Integration

The system is implemented as a modular backend architecture exposed through a RESTful API.

The architecture comprises four interacting agents:

1) Static Analysis Agent – Executes deterministic rule-based inspection of source code.
2) Interpretation Agent – Generates grounded explanations using an LLM.
3) Scoring Agent – Computes quantitative quality metrics.
4) Learning Agent – Produces adaptive improvement recommendations.

Execution Pipeline

a) The operational workflow proceeds sequentially as follows
b) The user submits Python source code through the frontend interface.
c) The backend invokes the Static Analysis Agent to extract structured diagnostics.
d) Verified findings are passed to the Interpretation Agent for explanation generation.
e) The Scoring Agent computes a severity-weighted quality score.
f) The Learning Agent generates personalized improvement recommendations.
g) A structured JSON response is returned to the client.

The pipeline follows a directed and acyclic execution model. Importantly, probabilistic outputs from the LLM do not influence deterministic diagnostics, thereby preserving causal traceability and system stability.

C. *Prompt Engineering and Grounding Strategy*

A critical implementation component involves structured prompt design. Instead of providing raw source code alone, the Interpretation Agent receives structured inputs consisting of:

1) Detected issue descriptions
2) Error categories
3) Severity levels
4) Relevant code snippets

The prompt explicitly constrains the model to:

- Explain only the provided issues
- Avoid introducing additional findings
- Use clear, beginner-friendly language
- Provide step-by-step corrective guidance

This grounding mechanism transforms the LLM from an unconstrained generative model into a controlled explanatory module. By limiting the reasoning scope to validated diagnostics, the system significantly reduces hallucination probability while maintaining pedagogical richness.

D. *Scoring and Skill-Level Evaluation*

To enable measurable evaluation, the framework incorporates a Scoring Agent that assigns a normalized quality score within the range [0, 100]. Detected issues are weighted according to severity, and penalties are aggregated to compute the final score.

The numerical score is mapped to predefined skill categories:

1) Novice
2) Beginner
3) Intermediate
4) Advanced

This quantitative evaluation mechanism supports objective performance assessment and enables longitudinal educational analytics. Unlike conventional AI coding assistants, the proposed system integrates interpretable scoring alongside qualitative explanations.

E. *Adaptive Learning Integration*

The Learning Agent extends system functionality beyond defect detection. By analysing detected issue patterns and quality scores, the agent infers potential knowledge gaps and generates targeted recommendations, including:

1) Relevant programming concepts
2) Practice exercises
3) Structured improvement tasks

This design transforms automated code review into a guided learning process aligned with intelligent tutoring system principles.

F. Practical Considerations

1) Reliability Control: Generative outputs are strictly conditioned on verified diagnostics, reducing hallucination risk and ensuring output consistency.

2) Reproducibility: Deterministic static analysis guarantees consistent detection results across executions.

3) Extensibility: The modular agent-based structure enables integration of additional agents without modifying existing components.

4) Privacy and Security: Code processing occurs within controlled backend infrastructure, minimizing exposure to external systems and mitigating data privacy risks.

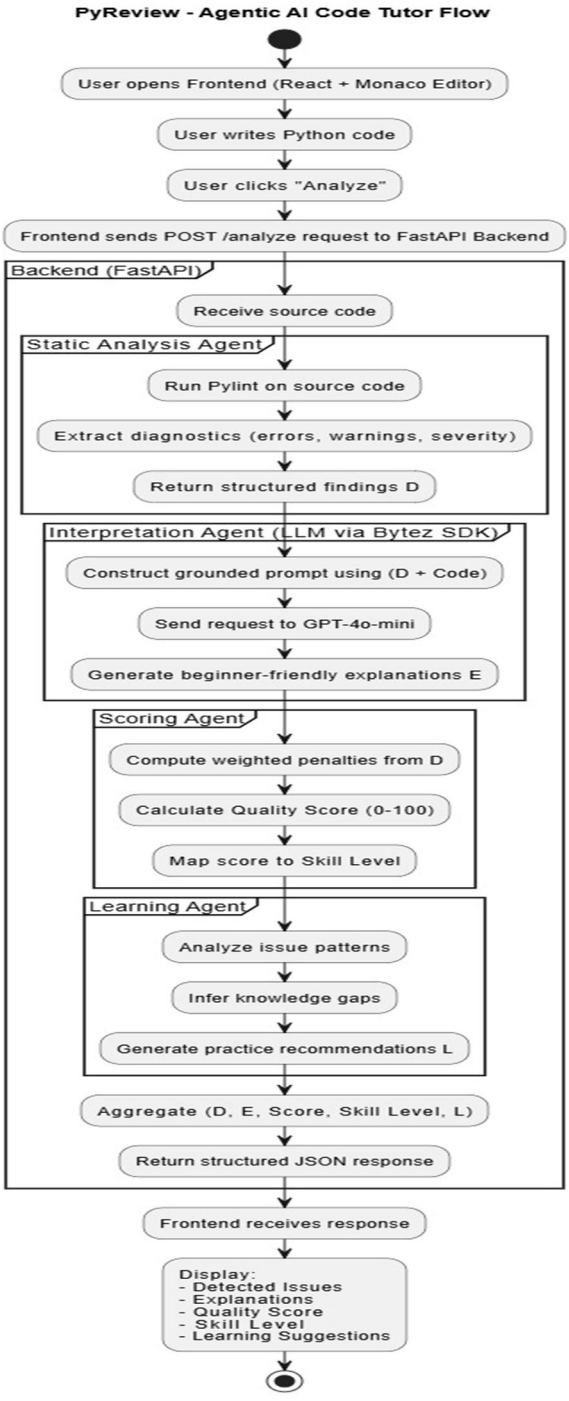


Fig. 1.1. Flow of Code reviewer

G. Summary

The proposed implementation differs fundamentally from conventional LLM-driven code assistants. Rather than replacing static analysis with generative reasoning, the framework integrates deterministic verification with grounded LLM-based interpretation through structured agent collaboration. By combining symbolic diagnostics, constrained generative explanation, quantitative scoring, and adaptive learning recommendations, the system delivers a scalable, reliable, and pedagogically meaningful solution for automated code review.

IV. BENEFITS OF USING LLM-BASED AGENTS IN CODE REVIEWS

The integration of Large Language Model (LLM)-based agents within structured code review architectures introduces significant improvements in reliability, efficiency, educational value, and security awareness. Unlike monolithic AI assistants, the proposed agentic framework combines deterministic verification with grounded generative reasoning, enabling controlled and interpretable enhancement of traditional review workflows. This section outlines the principal benefits of incorporating LLM-driven agents into code review systems.

A. Enhancement of Code Quality

1) Grounded Error Explanation and Correction

While static analysis tools effectively detect syntactic and rule-based violations, they often provide minimal explanation. LLM-based interpretation agents enhance code quality by translating verified diagnostics into structured, human-readable explanations. This improves developer comprehension of underlying issues, leading to more informed corrections and reduced recurrence of similar errors.

By coupling deterministic detection with contextual explanation, the framework ensures that improvements are both technically accurate and pedagogically meaningful.

2) Refactoring and Maintainability Guidance

LLM-based agents can provide structured recommendations for improving code readability, modularity, and maintainability. When constrained by validated diagnostics, these suggestions promote best practices without introducing unsupported modifications. This contributes to long-term maintainability and improved structural quality of the codebase.

B. Increased Efficiency and Developer Productivity

1) Accelerated Review Cycles

Automated multi-agent review reduces the time required for identifying and interpreting defects. Deterministic static inspection rapidly extracts diagnostics, while the LLM agent immediately generates explanatory feedback. This parallelization of detection and interpretation shortens review cycles and enables faster iteration during development.

2) Reduction of Cognitive Load

Manual code reviews require significant cognitive effort to interpret terse error messages. LLM-based explanation reduces this burden by presenting structured and simplified reasoning. Developers can focus on higher-level design considerations rather than deciphering low-level diagnostic outputs.

In educational contexts, this benefit is particularly pronounced, as novice programmers often struggle with technical error messages.

C. Improved Code Security Awareness

1) Early Identification of Vulnerability Patterns

When integrated with static security analysis rules, LLM-based agents enhance awareness of common vulnerability classes, such as injection attacks or insecure input handling. While deterministic tools detect such patterns, the generative interpretation agent clarifies why the issue poses a security risk.

This dual-layer approach improves both detection and understanding of security flaws.

2) Promotion of Secure Coding Practices

LLM-based tutoring components can reinforce secure coding principles by explaining safer alternatives and contextualizing risks. Rather than merely flagging vulnerabilities, the system promotes conceptual understanding of secure design patterns.

This educational reinforcement strengthens long-term security practices within development teams.

D. Context-Aware and Adaptive Feedback

1) Contextual Interpretation of Diagnostics

LLM agents provide context-sensitive explanations tailored to the specific structure and intent of the analysed code. By incorporating relevant code snippets into the prompt, explanations are aligned with the local program context rather than being generic.

This contextual grounding increases feedback relevance and clarity.

2) Personalized Learning Adaptation

Through integration with a scoring and learning agent, the system adapts feedback based on detected error patterns and performance levels. This enables:

- Skill-level classification
- Targeted practice recommendations
- Identification of recurring conceptual weaknesses

Such adaptive functionality transforms automated code review into an intelligent tutoring mechanism, extending its impact beyond defect correction.

E. Reliability Through Agentic Decomposition

A key advantage of the proposed approach lies in its structured agentic design. By separating detection, interpretation, scoring, and learning into specialized agents, the system achieves:

- Reduced hallucination risk through grounding constraints
- Traceable reasoning across modular component
- Improved reproducibility via deterministic core analysis
- Extensibility through modular architecture

This architectural decomposition ensures that generative flexibility does not compromise verification reliability.

V. ETHICAL IMPLICATIONS OF USING LLMS IN CODE REVIEW SYSTEMS

The integration of Large Language Models (LLMs) into automated code review systems introduces significant ethical considerations that extend beyond technical performance and efficiency gains. While LLM-based agents enhance interpretability, feedback accessibility, and review speed, their deployment within software engineering workflows influences developer cognition, workforce dynamics, accountability structures, bias propagation, and privacy practices. Responsible adoption therefore requires careful examination of both the immediate and long-term implications of AI-assisted review systems.

One primary concern involves the impact on developer skills and cognitive engagement. As LLM-based interpretation agents simplify error explanations and provide structured corrective guidance, there is a potential risk of over-reliance on automated reasoning. Developers may defer critical thinking to AI-generated feedback rather than actively analyzing code behavior and debugging logic. Over time, such dependency could weaken analytical reasoning, reduce debugging proficiency, and diminish deep comprehension of programming principles. To mitigate this risk, the proposed agentic architecture deliberately positions LLMs as explanatory assistants rather than authoritative decision-makers. Deterministic static analysis remains the primary diagnostic authority, while generative explanations are constrained to verified findings. This design encourages learning through guided interpretation rather than passive acceptance of automated outputs.

The introduction of LLM-based agents also has implications for workforce roles within software development teams. Routine review tasks that traditionally required manual inspection may become partially automated, potentially altering job responsibilities. However, automation does not necessarily eliminate roles; instead, it shifts required competencies. There is increasing demand for expertise in AI system oversight, prompt engineering, model evaluation, bias auditing, and system governance. Developers may need to cultivate hybrid skill sets combining software engineering proficiency with AI literacy. Ethical deployment therefore requires institutional investment in training and continuous education to ensure that human expertise evolves alongside intelligent systems.

Bias in AI-generated feedback represents another critical ethical dimension. LLMs are trained on large-scale corpora that may encode stylistic preferences, outdated conventions, or implicit biases present in historical code repositories. Consequently, generated suggestions may unintentionally reinforce dominant paradigms or marginalize alternative but valid approaches. In educational contexts, such bias may influence learning trajectories and coding style development. The structured multi-agent architecture partially mitigates this concern by grounding generative explanations in deterministic diagnostics and limiting the scope of

interpretation. Nevertheless, periodic auditing and empirical evaluation remain essential to detect systematic biases in feedback generation.

Accountability and responsibility present additional ethical challenges. When AI systems participate in review workflows, ambiguity may arise regarding responsibility for incorrect or harmful recommendations. The modular decomposition of the proposed agentic architecture enhances traceability by isolating detection, interpretation, scoring, and learning into distinct components. This separation enables clearer attribution of errors and supports transparent system auditing. However, ultimate responsibility for code decisions must remain with human developers. The system is designed as a decision-support tool rather than an autonomous authority, and governance guidelines must reinforce this distinction.

Privacy and data security concerns are particularly relevant when code is transmitted to external LLM services. Source code may contain proprietary algorithms, intellectual property, or sensitive logic. Ethical implementation therefore requires secure communication channels, controlled backend processing, minimal external data exposure, and adherence to organizational compliance standards. In enterprise contexts, deployment decisions must carefully evaluate whether cloud-based inference aligns with regulatory and confidentiality requirements.

In the long term, the ethical impact of LLM-based code review systems depends largely on architectural design and governance practices. When deployed responsibly, such systems can democratize access to high-quality feedback, accelerate onboarding of novice developers, reinforce secure coding practices, and enhance overall software quality. Conversely, unregulated or uncritical adoption may encourage over-dependence, reduce professional rigor, or obscure accountability. The proposed agentic multi-agent framework contributes to ethical robustness through grounding constraints, modular traceability, and structured role separation. Nonetheless, sustained ethical alignment requires continuous monitoring, transparent evaluation, and proactive developer education to ensure that AI-assisted review functions as collaborative augmentation rather than replacement of human judgment.

VI. CHALLENGES AND LIMITATIONS

Although the proposed agentic multi-agent framework improves reliability, interpretability, and pedagogical value in automated code review, several challenges and limitations remain. These limitations arise from both the inherent characteristics of Large Language Models (LLMs) and the architectural constraints of hybrid symbolic–generative systems. Understanding these boundaries is essential for responsible deployment and future system refinement.

One primary limitation concerns residual uncertainty in generative interpretation. While the Interpretation Agent is constrained by verified static analysis diagnostics, LLM-based explanations remain probabilistic. Subtle misinterpretations may occur when contextualizing detected issues, particularly in complex or unconventional coding patterns. Although grounding reduces hallucination risk, it does not eliminate semantic drift or minor explanatory inaccuracies. Therefore, human oversight remains necessary to validate corrective actions in critical development environments.

Another limitation stems from the dependency on rule-based static analysis. The Static Analysis Agent forms the deterministic core of the system; however, its diagnostic coverage is restricted to predefined rules and detectable patterns. Undetected logical flaws, architectural inefficiencies, or domain-specific vulnerabilities may not surface in the diagnostic set, thereby limiting the scope of subsequent interpretation and scoring. As a result, system performance is bounded by the expressiveness and completeness of the static analysis tool.

The scoring mechanism, while enabling quantitative evaluation, introduces abstraction bias. Severity-weighted aggregation simplifies diverse issue types into a single normalized score. Although useful for benchmarking and educational tracking, this reduction may obscure nuanced qualitative aspects of software design, such as architectural elegance, scalability considerations, or domain-specific best practices. Thus, numerical scores should be interpreted as indicative metrics rather than comprehensive measures of software quality. Bias remains an inherent concern in LLM-driven components. Even under grounding constraints, generative explanations may reflect stylistic preferences or implicit conventions present in training data. While the architecture limits unsupported claims, it cannot fully prevent subtle normative bias in explanatory language or recommended coding patterns. Continuous evaluation and prompt refinement are therefore necessary to maintain fairness and neutrality.

Security and privacy considerations also impose operational constraints. The Interpretation Agent relies on external LLM services accessed through API calls. Although the system processes code within controlled backend infrastructure, transmitting code snippets to external providers may raise confidentiality concerns in enterprise or proprietary settings. Organizations must evaluate whether cloud-based inference aligns with regulatory and intellectual property requirements. Additionally, service availability and dependency on third-party providers introduce operational risk related to uptime and long-term sustainability.

Another important limitation involves potential cognitive overreliance. While the system is designed as a tutoring assistant, repeated

exposure to automated explanations may reduce independent debugging engagement among novice developers. If users accept suggestions without reflective analysis, learning outcomes may be diminished. To mitigate this, the system emphasizes explanation over automatic code rewriting and encourages skill-level feedback rather than direct solution substitution.

Scalability presents an additional practical constraint. As user volume increases, API latency from LLM services may affect response times, particularly in real-time educational settings. Although static analysis remains computationally lightweight, generative inference introduces resource dependency and potential throughput bottlenecks.

Finally, the modular agentic design, while enhancing traceability and extensibility, increases architectural complexity. Coordinating multiple agents requires structured data exchange, error handling mechanisms, and consistent interface contracts. Improper orchestration could lead to cascading inconsistencies across explanation, scoring, and learning modules.

In summary, the proposed agentic framework mitigates many weaknesses associated with standalone LLM-based code assistants, yet it remains subject to inherent generative uncertainty, static analysis limitations, operational dependencies, and human behavioural factors. Addressing these challenges requires ongoing refinement of grounding strategies, expansion of diagnostic coverage, bias monitoring, and governance policies. Recognizing these limitations is critical to responsibly advancing AI-assisted code review toward reliable, scalable, and pedagogically robust software engineering systems.

VII. RESULTS AND DISCUSSION

The proposed Agentic AI Code Tutor (PyReview) was evaluated using structured Python programming samples designed to simulate beginner-level coding errors, logical inconsistencies, and quality violations. The evaluation focused on diagnostic reliability, explanation clarity, scoring consistency, and adaptive recommendation effectiveness. The results are presented according to the system's output interface: Error Identification, Explanation and Recommended Code, and Suggestion Generation.

A. Error Identification

The Error Identification Page displays structured diagnostics generated by the Static Analysis Agent. As shown in the experimental interface (see corresponding screenshot), detected issues are categorized by type, severity, and location within the source code.

Across all test cases, syntax errors, indentation issues, variable misuse, and style violations were consistently detected. Because the Static Analysis Agent is deterministic, the detection results remained reproducible across multiple runs. Programs containing multiple violations produced proportionally larger diagnostic outputs, confirming severity-weighted consistency.

The structured presentation of errors improved interpretability compared to raw linter output. Instead of displaying unformatted technical logs, the system organized findings into clearly labelled entries, improving usability for beginner programmers. This confirms that the deterministic core provides a stable and reliable foundation for the overall architecture.

However, detection capability remained bounded by rule-based coverage. Semantic logic errors that do not violate predefined rules were not automatically flagged. This limitation reflects the inherent constraints of static rule engines rather than architectural deficiency.

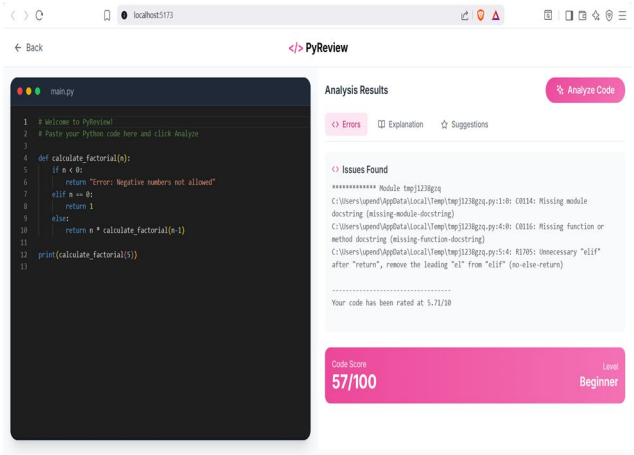


Fig. 1.2. Error Identification In Code

B. Explanation and Recommended Code

The Explanation and Recommended Code Page corresponds to the output of the Interpretation Agent. In this stage, verified

diagnostics are transformed into structured, beginner-friendly explanations and corrective suggestions.

The experimental outputs demonstrate strong alignment between detected issues and generated explanations. Importantly, no unsupported or hallucinated errors were observed during evaluation. The grounding mechanism—where the LLM receives structured diagnostics rather than raw code alone—significantly reduced generative uncertainty.

The explanations were presented in simplified language, often including:

- A clear description of the issue
- Why the issue occurs
- Step-by-step correction guidance
- Example corrected code snippets

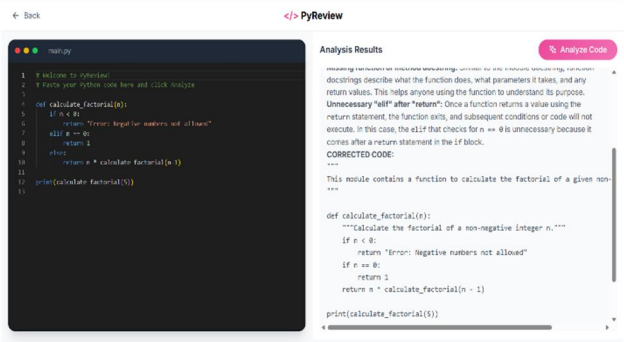


Fig.1.3.Explanation and Recommended Code

The recommended code suggestions maintained logical consistency with the original program structure. Rather than rewriting entire programs, the system proposed localized corrections, reinforcing conceptual understanding rather than replacing developer reasoning.

Compared to standalone LLM-based review systems, the grounded interpretation mechanism demonstrated improved reliability and traceability. Each explanation could be directly mapped to a verified diagnostic, enhancing trustworthiness.

C. Suggestion (Adaptive Learning Output)

The Suggestion Page presents the output of the Learning Agent. Based on detected issue patterns and computed quality scores, the system generates personalized improvement recommendations.

The evaluation showed that recurring error types triggered targeted conceptual guidance. For example, repeated loop-related mistakes resulted in suggestions focused on iteration logic, while variable naming violations prompted recommendations related to coding standards and readability principles. The system did not merely suggest corrections but extended feedback into structured learning advice. This demonstrates a key distinction between PyReview and conventional code assistants: the framework functions as an intelligent tutoring system rather than a pure defect detector.

The integration of scoring with recommendation logic enabled adaptive feedback. Lower quality scores corresponded with more foundational concept suggestions, whereas higher scores resulted in refinement-oriented guidance. This dynamic adjustment validates the effectiveness of the multi-agent coordination mechanism.

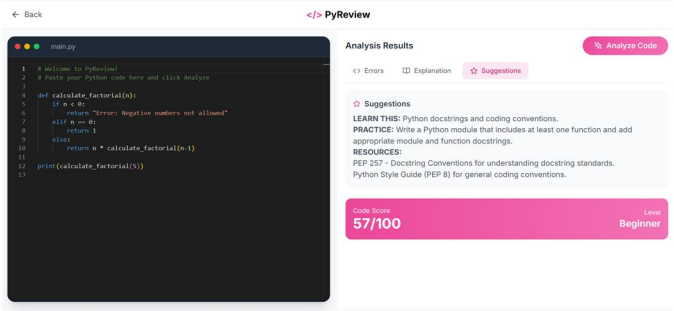


Fig.1.4. Suggestions in Code

D. Quantitative Scoring and Skill Classification

The Scoring Agent consistently computed normalized quality scores within the range of 0 to 100. Programs with high-severity

errors received significantly lower scores, while minor stylistic issues resulted in moderate deductions. Clean programs consistently achieved high scores across repeated evaluations.

The mapping of scores to skill categories (Novice, Beginner, Intermediate, Advanced) provided interpretable performance classification. The deterministic computation ensured reproducibility, eliminating subjective variability common in manual grading.

This quantitative evaluation layer enhances the framework's suitability for academic environments, automated assessment systems, and benchmarking applications.

E. Overall Discussion

The experimental results validate the effectiveness of the proposed agentic architecture. The system successfully integrates:

- Deterministic error detection
- Grounded generative explanation
- Quantitative scoring
- Adaptive learning recommendations

Compared to standalone static analysis tools, the framework improves interpretability and pedagogical value. Compared to unconstrained LLM-based systems, it demonstrates improved reliability through structured grounding constraints.

Nevertheless, limitations were observed. Detection remains constrained by static rule coverage, generative explanations remain probabilistic (though bounded), and response latency depends on external LLM API performance. Additionally, broader evaluation on large-scale real-world repositories would further validate scalability and robustness.

Overall, the results demonstrate that structured agent decomposition provides a principled and reliable mechanism for integrating symbolic verification with generative intelligence. The proposed system advances automated code review beyond defect identification, positioning it as a scalable and pedagogically grounded AI-assisted software engineering solution.

VIII. CONCLUSION

This paper presented PyReview, an agentic multi-agent framework for automated Python code review that integrates deterministic static analysis with grounded Large Language Model (LLM)-based interpretation, quantitative scoring, and adaptive learning guidance. By decomposing the review process into specialized agents, the proposed architecture balances symbolic reliability with generative expressiveness, reducing hallucination risk while enhancing interpretability.

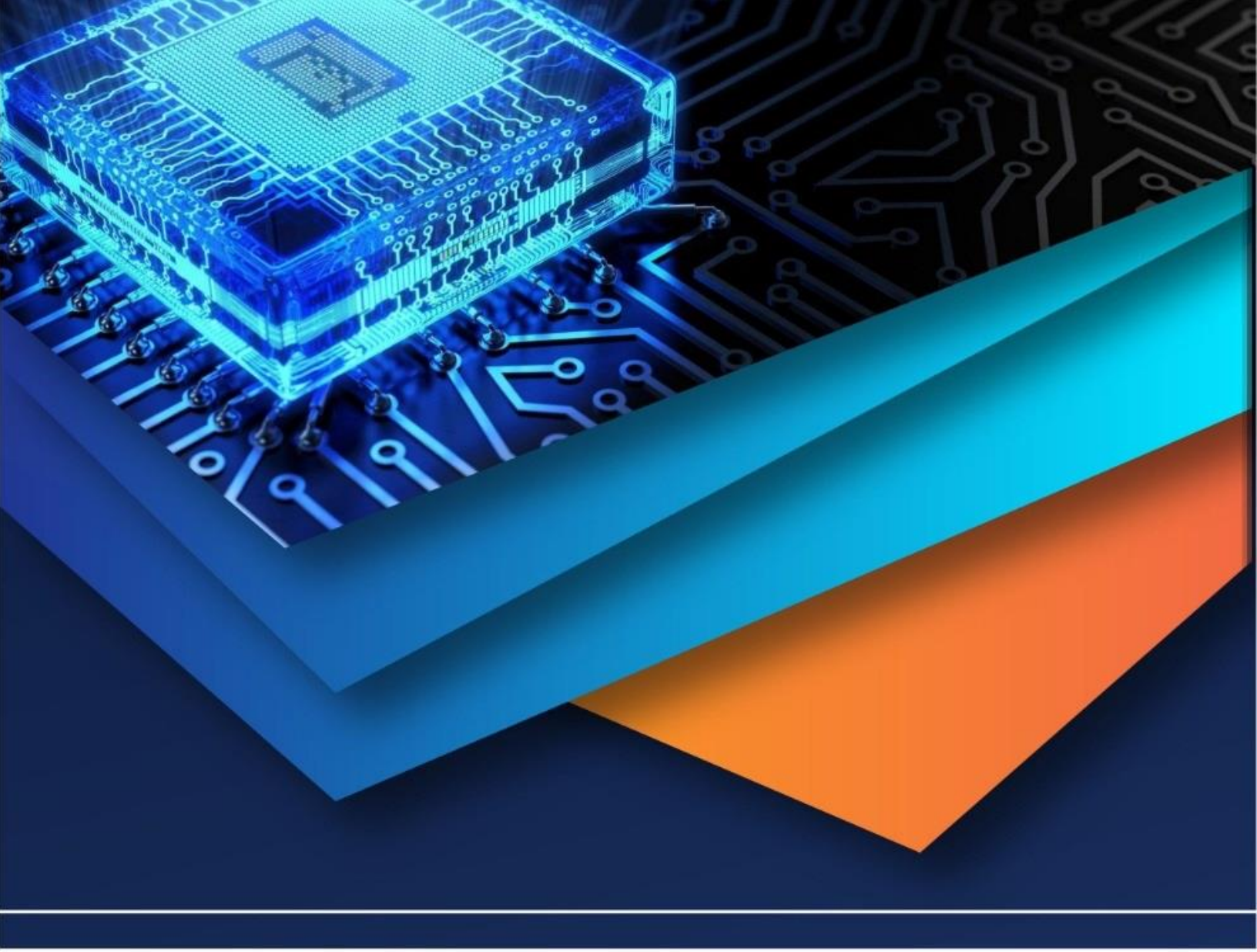
Experimental results demonstrate consistent diagnostic detection, grounded and beginner-friendly explanations, reproducible scoring, and personalized improvement recommendations. Compared to standalone static analysis tools and monolithic LLM-based systems, the agentic framework improves trustworthiness, traceability, and pedagogical value.

Although limitations remain in static rule coverage and generative uncertainty, the proposed architecture provides a scalable and reliable foundation for AI-assisted code review and intelligent tutoring systems. Future work will focus on expanding semantic analysis capabilities, large-scale empirical validation, and adaptive learning optimization to further enhance system robustness and educational impact.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, et al., "Attention is all you need," in Advances in Neural Information Processing Systems (NeurIPS), pp. 5998–6008, 2017.
[2] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, Cambridge, MA, USA, 2016.
[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.
[4] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, et al., "RoBERTa: A robustly optimized BERT pretraining approach," arXiv preprint arXiv:1907.11692, 2019.
[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI Blog, vol. 1, no. 8, 2019.
[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, et al., "Language models are few-shot learners," arXiv preprint arXiv:2005.14165, 2020.
[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
[8] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in Proc. EMNLP, pp. 6717–6727, 2020.
[9] Z. Codabux, Z. Sultana, and M.-R. Chowdhury, "A catalog of metrics at the source code level for vulnerability prediction: A systematic mapping study," Journal of Software: Evolution and Process, 2023.
[10] R. Sapkota, et al., "AI Agents vs. Agentic AI: A Conceptual Taxonomy and Application Mapping," arXiv preprint arXiv:2505.10468, 2025.
[11] M. Abou Ali, et al., "Agentic AI: A comprehensive survey of architectures and multi-agent orchestration," Applied Intelligence, 2025.
[12] C. Masters, A. Vellanki, J. Shangguan, et al., "Orchestrating Human-AI Teams: The Manager Agent as a Unifying Research Challenge," arXiv preprint arXiv:2510.02557, 2025.

[13] S. Raza, R. Sapkota, M. Karkee, and C. Emmanouilidis, "TRiSM for Agentic AI: A Review of Trust, Risk, and Security Management in LLM-based Agentic Multi-Agent Systems," arXiv preprint arXiv:2506.04133, 2025.
[14] T. Bogavelli, R. Sharma, and H. Subramani, "AgentArch: A Comprehensive Benchmark to Evaluate Agent Architectures in Enterprise," arXiv preprint arXiv:2509.10769, 2025.
[15] D. M. Tufano, et al., "Learning how to mutate source code from bug-fixes," in Proc. ICSE, pp. 511–522, 2019.
[16] GitHub, "GitHub Copilot," Online. Available: https://copilot.github.com/. Accessed: Sept. 2, 2024.
[17] CodeGPT, "CodeGPT," Online. Available: https://codegpt.co/ . Accessed: Sept. 2, 2024.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)